"Q+$\mathbb{Q}$"
Q Rational Number Library

Rob Hubbard

2005–6

Contents

This document is version (β)–316 for, and using, `rational.q` revision 309, `ratutils.q` revision 315, `rat_interval.q` revision 309 and Q 7.7.

It was generated from a pre-processed `rational.unpre.tex` revision 316 on June 10, 2007.

1 Introduction

1.1 The Rational Module

The module defines a type `Rational` for Albert Gräf's equational programming language 'Q' (`http://q-lang.sourceforge.net/`).
The module is compatible with Q version 7.7 (onwards).

1.2 The Files and the Default Prelude

The implementation of the `Rational` type and associated utilities is distributed across various files.

1.2.1 `rational.q` and Other Files

The file `rational.q` defines the type, its constructors and 'deconstructors' and basic arithmetical and mathematical operators and functions. This is included in the default prelude.

A few definitions associated with `Rational`s are defined in the appropriate file in the default prelude. For example: the type tests are contained in `typec.q`; the `floor` and `ceil` functions are taken care of in `math.q`; `min` and `max` are provided by `stdlib.q`.

It is also possible to create rational complex numbers (in addition to 'real' complex numbers and integral or Gaussian complex numbers). That is `Rational` plays nicely with `Complex`, as provided by Albert Gräf's `complex.q` in the default prelude. This is discussed further in §12.

1.2.2 `ratutils.q`

Additional '`Rational` utilities', not included in the default prelude, are defined in `ratutils.q`. The functions include further arithmetical and mathematical operators and functions, continued fraction support, approximation routines and string formatting and evaluation. Functions that require `ratutils.q` to be explicitly included are indicated like `this`$^{(\mathcal{U})}$.

The `Rational` utilities include some 'rational complex number' functions.

1.2.3 `rat_interval.q`

Amongst the `Rational` utilities are some functions that return a rational interval. The file `rat_interval.q` is a partial implementation of interval arithmetic, and is not included in the default prelude. Intervals are discussed further in §9.1.

1.3 Notation

Throughout this document, the parameters Q, Q_0, Q_1,... usually denote `Rational`s ($\in \mathbb{Q}$), parameters Z,... usually denote integers ($\in \mathbb{Z}$; `Int`s), R,... usually denote real numbers ($\in \mathbb{R}$; `Real`s), C,... usually denote complex numbers ($\in \mathbb{C}$), N,... usually denote parameters of any numeric type, V,... usually denote parameters of any interval type, and X,... usually denote parameters of any type.

The `Real`s are not just the `Float`s, but include `Rational`s and `Int`s.

The term 'rational' usually refers to a rational number $\in \mathbb{Q} \supset \mathbb{Z}$, or an expression of type `Rational` or `Int`.

Functions that require `ratutils.q` to be explicitly included are indicated like `this`$^{(\mathcal{U})}$.

1.4 Acknowledgements

Thank you to Dr Albert Gräf for helpful feedback on the Q language and for answering my many questions. Albert performed the organisational work, splitting my original main source file for partial inclusion in the default prelude, and provided some of the 'architectural' functions required for smooth and consistent integration with his Q system.

Thanks to various members of the Q users mailing list (https://lists.sourceforge.net/lists/listinfo/q-lang-users) for the feedback and suggestions.

2 The `Rational` Type

2.1 Constructors

`Rational`s are constructed with the function `rational`.

`rational` (Z_1, Z_2) — given a pair of integers (Z_1, Z_2), this returns the `Rational` equivalent to the fraction Z_1/Z_2.

This is the inverse (up to equivalence) of `num_den` (see §2.2).

Example 1 *Constructing a fraction.*

```
==> rational (44,14)
22%7
```

`rational` Z — given an integer Z, this returns the `Rational` equivalent to the `Int` Z.

Example 2 *Converting from an* `Int`.

```
==> rational 3
3%1
```

`(%)` N_1 N_2 — is a `Rational`-aware division function, which may be used as a constructor (for `Int`s N_1 and N_2). This is described in §3.2.

2.2 'Deconstructors'

A rational number is in simplest form if the numerator and denominator are coprime (i.e. do not have a factor in common) and the denominator is positive (and, specifically, non-zero). Sometimes the term 'irreducible' is used for a rational in simplest form. This is a property of the representation of the rational number and not of the number itself.

`num` Q — given a `Rational` or `Int` Q, returns the '(signed) simplest numerator', i.e. the numerator of the normalised form of Q.

`den` Q — given a `Rational` or `Int` Q, returns the '(positive) simplest denominator', i.e. the denominator of the normalised form of Q.

`num_den` Q — given a `Rational` or `Int` Q, returns a pair (N, D) containing the (signed) simplest numerator N and the (positive) simplest denominator D.

This is the inverse (up to equivalence) of `rational` as defined on integer pairs (see §2.1).

Example 3 *Using num_den to obtain a representation in simplest form.*

```
==> def Q = (44%(-14))
==> num Q; den Q
-22
7
==> num_den Q
(-22,7)
==> num_den 3
(3,1)
==> num_den (-3)
(-3,1)
```

Together, `num` and `den` are a pair of 'decomposition' operators, and `num_den` is also a decomposition operator. There are others (see §10).

The `integer_and_fraction` function (see §6.2) may be used in conjunction with `num_den` to decompose a `Rational` into integer, numerator and denominator parts.

2.3 Type Predicate

This type may be used as a type predicate in the usual way (`Q:Rational`).

2.4 Type and Value Tests

The functions `isrational` and `isratval` and other rational variants are new for `Rationals` and the standard functions `isexact` and `isinexact` are extended for `Rationals`.

A value is 'exact', or of an exact type, if it is of a type that is able to represent the values returned by arithmetical operations exactly; in a sense, it is 'closed' under arithmetical operations. Otherwise, a value is 'inexact'. Inexact types are able to store some values only approximately.

`Float` is not an exact type. The results of some operations on some values that are stored exactly, can't be stored exactly. (Furthermore, `Floats` are intended to represent real numbers; no irrational number ($\in \mathbb{R} \setminus \mathbb{Q}$) can be stored exactly as a `Float`; even some rational ($\in \mathbb{Q}$) numbers are not stored exactly.)

`Rational` is an exact type. All rational numbers (subject to available resources, of course) are stored exactly. The results of the arithmetical operations on `Rationals` are `Rationals` represented exactly.

Beware that the standard `isintval` and `isratval` may return `true` even if the value is of `Float` type. However, these functions may be combined with `isexact`.

`isexact` X — returns whether X has an exact value.

`isinexact` X — returns whether X has an inexact value.

`isrational` X — returns whether X is of type `Rational`.

`isratval` X — returns whether X has a rational value.

Example 4 *Rational value tests.*

```
==> def L = [9, 9%1, 9%2, 4.5, sqrt 2, 1+i, inf, nan]
==> map isexact L
[true,true,true,false,false,true,false,false]
==> map isinexact L
[false,false,false,true,true,false,true,true]
==> map isrational L
[false,true,true,false,false,false,false,false]
==> map isratval L
[true,true,true,true,true,false,false,false]
==> map (lambda X (isexact X and isratval X)) L // "has exact rational value"
[true,true,true,false,false,false,false,false]
==> map isintval L // for comparison
[true,true,false,false,false,false,false,false]
==> map (lambda X (isexact X and isintval X)) L // "has exact integer value"
[true,true,false,false,false,false,false,false]
```

See §12 for details about rational complex numbers, and §12.2 for details of their type and value tests.

2.5 Internal Representation

It is not appropriate to give details of the internal representation of rational numbers as implemented by this module. That is subject to change.

The module makes no *guarantee* as to whether the internal representation is in simplest form, how the sign is handled, nor what components are stored (integer part, sign, numerator and denominator of fraction part,...).

The representation is `private`, therefore you can not operate on rationals except through the public interface.

The representation is also normally hidden by the Q system through the standard special `view` function.

However, it is useful to be able to understand responses given by the module when debugging.

The value "`rational::rat` N D" (or just "`rat` N D") in the current implementation may be interpreted as the rational number $\frac{N}{D}$.

See, for example, §2.2 for details of a public interface function providing a representation with guaranteed properties.

3 Arithmetic

3.1 Operators

The standard arithmetic operators $(+)$, $(-)$ and $(*)$ are overloaded to have at least one `Rational` operand. If both operands are `Rational` then the result is `Rational`. If one operand is `Int`, then the result is `Rational`. If one operand is `Float`, then the result is `Float`.

The operators (/) and (%) are overloaded for division on at least one `Rational` operand. The value returned by (/) is always `inexact` (in the sense of §2.4). The value returned by (%) is `exact` (if it exists).

The standard function `pow` is overloaded to have a `Rational` left operand. If `pow` is passed `Int` operands where the right operand is negative, then a `Rational` is returned. The right operand should be an `Int`; negative values are permitted (because $Q^{-Z} = 1/Q^Z$). It is not overloaded to also have a `Rational` right operand because such values are not generally rational (e.g. $Q^{\frac{1}{N}} = \sqrt[N]{Q}$).

The standard arithmetic operator (^) is also overloaded, but produces a `Float` value (as always).

The values of `pow 0 0` and 0^0 (with `Int` or `Rational` zeroes) are left undefined.

Example 5 *Arithmetic.*

```
==> 5%7 + 2%3 ; str_mixed _
29%21
"1+8/21"
==> 1 + 2%3
5%3
==> _ + 1.0
2.66666666666667
==> 3%8 - 1%3
1%24
==> (11%10) ^ 3
1.331
==> pow (11%10) 3
1331%1000
==> pow 3 5
243
==> pow 3 (-5)
1%243
```

(The function `str_mixed` *is described in §14.1.)*

Beware that (/) on `Int`s will not produce a `Rational` result.

Example 6 *Division.*

```
==> 44/14
3.14285714285714
==> 44%14
22%7
==> str_mixed _
"3+1/7"
```

(The function `str_mixed` *is described in §14.1.)*

3.2 More on Division

There is a `Rational`-aware divide operator on the numeric types:

N_1 `%` N_2 — returns the quotient ($\in \mathbb{Q}$) of N_1 and N_2. If N_1 and N_2 are `Rational` or `Int` then the result is `Rational`. This operator has the precedence of division (`/`).

Example 7 *Using* `%` *like a constructor.*

```
==> 44 % 14
22%7
==> 2 + 3%8 // "2 3/8"
19%8
==> str_mixed _
"2+3/8"
```

(The function `str_mixed` *is described in §14.1.)*

`reciprocal`$^{(\mathcal{U})}$ N — returns the reciprocal of N: $1/N$.

Example 8 *Reciprocal.*

```
==> reciprocal (22%7)
7%22
```

The following division functions are parameterised by a rounding mode *Round*. The available rounding modes are described in §6.1.

`divide`$^{(\mathcal{U})}$ *Round* N D — for `Rationals` N and D returns a pair (Q, R) of 'quotient' and 'remainder' where Q is an `Int` and R is a `Rational` such that $|R| < |D|$ (or better) and $N = Q * D + R$. Further conditions may hold, depending on the chosen rounding mode *Round* (see §6.1).

If *Round* = `floor` then $0 \leq R < D$. If *Round* = `ceil` then $-D < R \leq 0$.

If *Round* = `trunc` then $|R| < |D|$ and $\operatorname{sgn} R \in \{0, \operatorname{sgn} D\}$.

If *Round* = `round`, *Round* = `round_zero_bias` or *Round* = `round_unbiassed` then $|R| \leq D/2$.

`quotient`$^{(\mathcal{U})}$ *Round* N D — returns just the quotient as produced by `divide` *Round* N D.

`modulus`$^{(\mathcal{U})}$ *Round* N D — returns just the remainder as produced by `divide` *Round* N D.

Q_1 `div`$^{(\mathcal{U})}$ Q_2 — (overload of the built-in `div`) Q_1 and Q_2 may be `Rational` or `Int`. Returns an `Int`.

Q_1 `mod`$^{(\mathcal{U})}$ Q_2 — (overload of the built-in `mod`) Q_1 and Q_2 may be `Rational` or `Int`. Returns a `Rational`.

If $Q = Q_1$ `div` Q_2 and $R = Q_1$ `mod` Q_2 then $Q_1 = Q * Q_2 + R$, $Q \in \mathbb{Z}$, $|R| < |Q_2|$ and $\operatorname{sgn} R \in \{0, \operatorname{sgn} Q_2\}$.

3.3 Relations — Equality and Inequality Tests

The standard arithmetic operators (=), (<>), (<), (<=), (>), (>=) are overloaded to have at least one `Rational` operand. The other operand may be `Rational`, `Int` or `Float`.

Example 9 *Inequality.*

```
==> 3%8 < 1%3
false
```

3.4 Comparison Function

`cmp`$^{(u)}$ N_1 N_2 is the ‘comparison’ (or ‘compare’) function, and returns sgn $(N_1 - N_2)$; that is, it returns -1 if $N_1 < N_2$, 0 if $N_1 = N_2$, and $+1$ if $N_1 > N_2$.

Example 10 *Compare.*

```
==> cmp (3%8) (1%3)
1
```

4 Mathematical Functions

Most mathematical functions, including the elementary functions (sin, $\sin^{-1}$, sinh, $\sinh^{-1}$, cos,..., exp, ln,...), are not closed on the set of rational numbers. That is, most mathematical functions do not yield a rational number in general when applied to a rational number. Therefore the elementary functions are not defined for `Rational`s. To apply these functions, first apply a cast to `Float`, or compose the function with a cast.

4.1 Absolute Value and Sign

The standard `abs` and `sgn` functions are overloaded for `Rational`s.

`abs` Q — returns absolute value, or magnitude, $|Q|$ of Q; `abs` $Q = |Q| = Q \times \text{sgn}\ Q$ (see below).

`sgn` Q — returns the sign of `Q` as an `Int`; returns -1 if $Q < 0$, 0 if $Q = 0$, $+1$ if $Q > 0$.

Together, these functions satisfy the property $\forall Q \bullet (\text{sgn}\ Q) * (\text{abs}\ Q) = Q$ (i.e. $\forall Q \bullet (\text{sgn}\ Q) * |Q| = Q$). Thus these provide a pair of ‘decomposition’ operators; there are others (see §10).

4.2 Greatest Common Divisor (GCD) and Least Common Multiple (LCM)

The standard functions `gcd` and `lcm` are overloaded for `Rational`s, and mixtures of `Int` and `Rational`.

`gcd`$^{(u)}$ N_1 N_2 — The GCD is also known as the Highest Common Factor (HCF). The GCD of rationals Q_1 and Q_2 is the largest (therefore positive) rational F such that F divides into both Q_1 and Q_2 exactly, i.e. an integral number of times.

This is not defined for N_1 and N_2 both zero.

For integral Q_1 and Q_2, this definition coincides with the usual definition of GCD for integers.

Example 11 *With two* `Rational`*s.*

```
==> def A = 7%12; def B = 21%32
==> def F = gcd A B ; F
7%96
==> A % F
8%1
==> B % F
9%1
```

Example 12 *With a* `Rational` *and an* `Int`*.*

```
==> def F = gcd (6%5) 4 ; F
2%5
==> (6%5) % F
3%1
==> 4 % F
10%1
```

Example 13 *With integral* `Rational`*s and with* `Int`*s.*

```
==> gcd (rational 18) (rational 24)
6%1
==> gcd 18 24
6
```

Example 14 *The behaviour with negative numbers.*

```
==> gcd (rational (-18)) (rational 24)
6%1
==> gcd (rational 18) (rational (-24))
6%1
==> gcd (rational (-18)) (rational (-24))
6%1
```

`lcm`$^{(u)}$ N_1 N_2 — The LCM of rationals Q_1 and Q_2 is the smallest positive rational M such that both Q_1 and Q_2 divide M exactly.

This is not defined for N_1 and N_2 both zero.

For integral Q_1 and Q_2, this definition coincides with the usual definition of LCM for integers.

Example 15 *With two* `Rational`*s.*

```
==> def A = 7%12; def B = 21%32
==> def M = lcm A B ; M
21%4
==> M % A
9%1
==> M % B
8%1
```

Example 16 *With a* `Rational` *and an* `Int`*.*

```
==> def M = lcm (6%5) 4 ; M
12%1
==> M % (6%5)
10%1
```

Example 17 *The behaviour with negative numbers.*

```
==> lcm (rational (-18)) (rational 24)
72%1
==> lcm (rational 18) (rational (-24))
72%1
==> lcm (rational (-18)) (rational (-24))
72%1
```

Together, the GCD and LCM have the following property when applied to **two** numbers:
(gcd Q_1 Q_2) $\times$ (lcm Q_1 Q_2) $= |Q_1 \times Q_2|$.

4.3 Extrema (Minima and Maxima)

The standard `min` and `max` functions work with `Rational` values.

Example 18 *Maximum.*

```
==> max (3%8) (1%3)
3%8
```

5 Special Rational Functions

5.1 Complexity

The ‘complexity’ (or ‘complicatedness’) of a `Rational` is a measure of the greatness of its simplest (positive) denominator.

The complexity of a number is not itself made available, but various functions and operators are provided to allow complexities to be compared. Generally, it does not make sense to operate directly on complexity values.

The complexity functions in this section may be applied to `Int`s (the least complex), `Rational`s, or reals (`Float`s; the most complex).

Functions concerning ‘complexity’ are named with ‘`cplx`’, whereas functions concerning ‘complex numbers’ (see §12) are named with ‘`comp`’.

5.1.1 Complexity Relations

N_1 `eq_cplx`$^{(\mathcal{U})}$ N_2 — “[is] equally complex [to]” — returns `true` if N_1 and N_2 are equally complex; returns `false` otherwise. Equal complexity is not the same a equality; N_1 and N_2 are equally complex if their simplest denominators are equal. Equal complexity forms an equivalence relation on `Rational`s.

Example 19 *Complexity equality test.*

```
==> (1%3) eq_cplx (100%3)
true
==> (1%4) eq_cplx (1%5)
false
==> (3%3) eq_cplx (1%3) // LHS is not in simplest form
false
```

N_1 `not_eq_cplx`$^{(\mathcal{U})}$ N_2 — “not equally complex” — returns `false` if N_1 and N_2 are equally complex; returns `true` otherwise.

N_1 `less_cplx`$^{(\mathcal{U})}$ N_2 — “[is] less complex [than]” (or “simpler”) — returns `true` if N_1 is strictly less complex than N_2; returns `false` otherwise. This forms a partial strict ordering on `Rational`s.

Example 20 *Complexity inequality test.*

```
==> (1%3) less_cplx (100%3)
false
==> (1%4) less_cplx (1%5)
true
==> (3%3) less_cplx (1%3) // LHS is not in simplest form
true
```

N_1 `less_eq_cplx`$^{(\mathcal{U})}$ N_2 — “less or equally complex” (or “not more complex”) — returns `true` if N_1 is less complex than or equally complex to N_2; returns `false` otherwise. This forms a partial non-strict ordering on `Rational`s.

N_1 `more_cplx`$^{(\mathcal{U})}$ N_2 — “[is] more complex [than]” — returns `true` if N_1 is strictly more complex than N_2; returns `false` otherwise. This forms a partial strict ordering on `Rational`s.

N_1 `more_eq_cplx`$^{(\mathcal{U})}$ N_2 — “more or equally complex” (or “not less complex”) — returns `true` if N_1 is more complex than or equally complex to N_2; returns `false` otherwise. This forms a partial non-strict ordering on `Rational`s.

5.1.2 Complexity Comparison Function

`cmp_complexity`$^{(\mathcal{U})}$ N_1 N_2 — is the 'complexity comparison' function, and returns the sign of the difference in complexity; that is, it returns -1 if N_1 is less complex than N_2, 0 if N_1 and N_2 are equally complex (but not necessarily equal), and $+1$ if N_1 is more complex than N_2.

Example 21 *Complexity comparison.*

```
==> cmp_complexity (1%3) (100%3)
0
==> cmp_complexity (1%4) (1%5)
-1
==> cmp_complexity (3%3) (1%3) // LHS is not in simplest form
-1
```

5.1.3 Complexity Extrema

`least_cplx`$^{(\mathcal{U})}$ N_1 N_2 — returns the least complex of N_1 and N_2; if they're equally complex, N_1 is returned.

Example 22 *Complexity selection.*

```
==> least_cplx (100%3) (1%3)
100%3
==> least_cplx (1%5) (1%4)
1%4
==> least_cplx (1%3) (3%3) // second argument not in simplest form
1%1
```

`most_cplx`$^{(\mathcal{U})}$ N_1 N_2 — returns the most complex of N_1 and N_2; if they're equally complex, N_1 is returned.

5.1.4 Other Complexity Functions

`complexity_rel`$^{(\mathcal{U})}$ N_1 Op N_2 — returns "complexity-of N_1" compared by operator Op to the "complexity-of N_2". This is equivalent to `prefix_complexity_rel` Op N_1 N_2 (below), but is the more readable form.

Example 23 *Complexity relations.*

```
==> complexity_rel (1%3) (=) (100%3)
true
==> complexity_rel (1%4) (<=) (1%5)
true
==> complexity_rel (1%4) (>) (1%5)
false
```

`prefix_complexity_rel`$^{(\mathcal{U})}$ Op N_1 N_2 — returns the same as `complexity_rel` N_1 Op N_2, but this form is more convenient for currying.

5.2 Mediants and Farey Sequences

`mediant`$^{(\mathcal{U})}$ Q_1 Q_2 — returns the canonical mediant of the rationals Q_1 and Q_2, a form of (non-arithmetic) average on rationals. The mediant of the representations $N_1/D_1 = Q_1$ and $N_2/D_2 = Q_2$, where D_1 and D_2 must be positive, is defined as $(N_1 + N_2)/(D_1 + D_2)$. A mediant of the rationals Q_1 and Q_2 is a mediant of some representation of each of Q_1 and Q_2. That is, the mediant is dependent upon the representations and therefore is not well-defined as a function on pairs of rationals. The canonical mediant always assumes the simplest representation, and therefore is well-defined as a function on pairs of rationals.

By the phrase “the mediant” (as opposed to just “a mediant”) we always mean “the canonical mediant”.

If $Q_1 < Q_2$, then any mediant Q is always such that $Q_1 < Q < Q_2$.

The (canonical) mediant has some special properties. If Q_1 and Q_2 are integers, then the mediant is the arithmetic mean. If Q_1 and Q_2 are unit fractions (reciprocals of integers), then the mediant is the harmonic mean. The mediant of Q and $1/Q$ is ± 1, (which happens to be a geometric mean with the correct sign, although this is a somewhat uninteresting and degenerate case).

Example 24 *Mediants.*

```
==> mediant (1%4) (3%10)
2%7
==> mediant 3 7 // both Ints
5%1
==> mediant 3 8 // both Ints again
11%2
==> mediant (1%3) (1%7) // both unit fractions
1%5
==> mediant (1%3) (1%8) // both unit fractions again
2%11
==> mediant (-10) (-1%10)
-1%1
```

`farey`$^{(\mathcal{U})}$ K — for an `Int` K, `farey` returns the ordered list containing the order-K Farey sequence, which is the ordered list of all rational numbers between 0 and 1 inclusive with (simplest) denominator at most K.

Example 25 *A Farey sequence.*

```
==> map str_mixed (farey 6)
["0","1/6","1/5","1/4","1/3","2/5","1/2","3/5","2/3","3/4","4/5","5/6","1"]
```

(The function `str_mixed` *is described in §14.1.)*

Farey sequences and mediants are closely related. Three rationals $Q_1 < Q_2 < Q_3$ are consecutive members of a Farey sequence if and only if Q_2 is the mediant of Q_1 and Q_3.

If rationals $Q_1 = N_1/D_1 < Q_2 = N_2/D_2$ are consecutive members of a Farey sequence, then $N_2 D_1 - N_1 D_2 = 1$.

5.3 Rational Type Simplification

`rat_simplify`$^{(\mathcal{U})}$ Q — returns Q with `Rational`s simplified to `Int`s, if possible.

Example 26 *Rational type simplification.*

```
==> def L = [9, 9%1, 9%2, 4.5, 9%1+i, 9%2+i]; L
[9,9%1,9%2,4.5,9%1+:1,9%2+:1]
==> map rat_simplify L
[9,9,9%2,4.5,9+:1,9%2+:1]
```

See §12 for details about rational complex numbers, and §12.5 for details of their type simplification.

6 $\mathbb{Q} \to \mathbb{Z}$ — Rounding

6.1 Rounding to Integer

Some of these are new functions, and some are overloads of standard functions. The behaviour of the overloads is consistent with that of the standard functions.

`floor` Q — (overload of standard function introduced in Q 7.1) returns Q rounded downwards, i.e. towards $-\infty$, to an `Int`, usually denoted $\lfloor Q \rfloor$.

`ceil` Q — (overload of standard function introduced in Q 7.1) returns Q rounded upwards, i.e. towards $+\infty$, to an `Int`, usually denoted $\lceil Q \rceil$.

`trunc` Q — (overload of standard function) returns Q truncated, i.e. rounded towards 0, to an `Int`.

`round` Q — (overload of standard function) returns Q 'rounded off', i.e. rounded to the nearest `Int`, with 'half-integers' (values that are an integer plus a half) rounded away from zero.

`round_zero_bias`$^{(\mathcal{U})}$ Q — (new function) returns Q 'rounded off', i.e. rounded to the nearest `Int`, but with 'half-integers' rounded towards zero.

`round_unbiased`$^{(\mathcal{U})}$ Q — (new function) returns Q rounded to the nearest `Int`, with 'half-integers' rounded to the nearest even `Int`.

Example 27 *Illustration of the different rounding modes.*

```
==> def L = while (<= 3) (+(1%2)) (- rational 3)
==> map float L // (just to show the values in a familiar format)
[-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0]
==> map floor L
[-3,-3,-2,-2,-1,-1,0,0,1,1,2,2,3]
==> map ceil L
[-3,-2,-2,-1,-1,0,0,1,1,2,2,3,3]
==> map trunc L
[-3,-2,-2,-1,-1,0,0,0,1,1,2,2,3]
==> map round L
[-3,-3,-2,-2,-1,-1,0,1,1,2,2,3,3]
==> map round_zero_bias L
[-3,-2,-2,-1,-1,0,0,0,1,1,2,2,3]
==> map round_unbiased L
[-3,-2,-2,-2,-1,0,0,0,1,2,2,2,3]
```

(The function `float` *is described in §8.)*

6.2 Integer and Fraction Parts

`integer_and_fraction`$^{(\mathcal{U})}$ *Round* Q — returns a pair (Z, F) where Z is the 'integer part' as an `Int`, F is the 'fraction part' as a `Rational`, where the rounding operations are performed using rounding mode *Round* (see §6.1).

Example 28 *Integer and fraction parts with the different rounding modes.*

```
==> def NC = -22%7
==> integer_and_fraction floor NC
(-4,6%7)
==> integer_and_fraction trunc NC
(-3,-1%7)
==> integer_and_fraction round NC
(-3,-1%7)
```

It is always the case that Z and F have the property that $Q = Z + F$. However, the remaining properties depend upon the choice of *Round*.

Thus this provides a 'decomposition' operator; there are others (see §10).

If *Round* = `floor` then $0 \leq F < 1$. If *Round* = `ceil` then $-1 < F \leq 0$.

If *Round* = `trunc` then $|F| < 1$ and $\text{sgn}\ F \in \{0, \text{sgn}\ Q\}$.

If *Round* = `round`, *Round* = `round_zero_bias` or *Round* = `round_unbiassed` then $|F| \leq 1/2$.

`fraction`$^{(\mathcal{U})}$ *Round Q* — returns just the 'fraction part' as a `Rational`, where the rounding operations are performed using *Round*.

The corresponding function '`integer`' is not provided, as `integer` *Round Q* would be just *Round Q*.

The `integer_and_fraction` function (probably with `trunc` or `floor` rounding mode) may be used in conjunction with `num_den` (see §2.2) to decompose a `Rational` into integer, numerator and denominator parts.

`int` Q — overloads the built-in `int` and returns the 'integer part' of Q consistent with the built-in.

`frac` Q — overloads the built-in `frac` and returns the 'fraction part' of Q consistent with the built-in.

Example 29 *Standard integer and fraction parts.*

```
==> def NC = -22%7
==> int NC
-3.0
==> frac NC
-0.142857142857143
```

7 Rounding to Multiples

`round_to_multiple`$^{(\mathcal{U})}$ *Round MultOf Q* — returns Q rounded to an integer multiple of a non-zero value $MultOf$, using *Round* as the rounding mode (see §6.1). Note that it is the multiple that is rounded in the prescribed way, and not the final result, which may make a difference in the case that $MultOf$ is negative. If that is not the desired behaviour, pass this function the absolute value of $MultOf$ rather than $MultOf$. Similar comments apply to the following functions.

`floor_multiple`$^{(\mathcal{U})}$ $MultOf\ Q$ — returns Q rounded to a downwards integer multiple of $MultOf$.

`ceil_multiple`$^{(\mathcal{U})}$ $MultOf\ Q$ — returns Q rounded to an upwards integer multiple of $MultOf$.

`trunc_multiple`$^{(\mathcal{U})}$ $MultOf\ Q$ — returns Q rounded towards zero to an integer multiple of $MultOf$.

`round_multiple`$^{(\mathcal{U})}$ $MultOf\ Q$ — returns Q rounded towards the nearest integer multiple of $MultOf$, with half-integer multiples rounded away from 0.

`round_multiple_zero_bias`$^{(\mathcal{U})}$ $MultOf\ Q$ — returns Q rounded towards the nearest integer multiple of $MultOf$, with half-integer multiples rounded towards 0.

`round_multiple_unbiased`$^{(\mathcal{U})}$ $MultOf\ Q$ — returns Q rounded towards the nearest integer multiple of $MultOf$, with half-integer multiples rounded to an even multiple.

Example 30 *Round to multiple.*

```
==> def L = [34.9, 35, 35%1, 35.0, 35.1]
==> map float L // (just to show the values in a familiar format)
[34.9,35.0,35.0,35.0,35.1]
==> map (floor_multiple 10) L
[30,30,30,30,30]
==> map (ceil_multiple 10) L
```

```
[40,40,40,40,40]
==> map (trunc_multiple 10) L
[30,30,30,30,30]
==> map (round_multiple 10) L
[30,40,40,40,40]
==> map (round_multiple_zero_bias 10) L
[30,30,30,30,40]
==> map (round_multiple_unbiased 10) L
[30,40,40,40,40]
```

(The function `float` *is described in §8.)*

The round_multiple functions may be used to find a fixed denominator approximation of a number. (The simplest denominator may actually be a proper factor of the chosen value.)

To approximate for a bounded (rather than particular) denominator, use `rational_approx_max_den` instead (see §9.3).

Example 31 *Finding the nearest* $Q = N/D$ *value to* $1/e \approx 0.368$ *where* $D = 1000$ *(actually, where* $D|1000$*).*

```
==> def Co_E = exp (-1) ; Co_E
0.367879441171442
==> round_multiple (1%1000) Co_E
46%125
==> 1000 * _
368%1
```

Example 32 *Finding the nearest* $Q = N/D$ *value to* $1/\phi \approx 0.618$ *where* $D = 3^5 = 243$ *(actually, where* $D|243$*).*

```
==> def Co_Phi = (sqrt 5 - 1) / 2
==> round_multiple (1%243) Co_Phi
50%81
```

Other methods for obtaining a `Rational` approximation of a number are described in §9

8 $\mathbb{Q} \to \mathbb{R}$ — Conversion / Casting

`float` Q — (overload of built-in) returns a `Float` having a value as close as possible to Q.

(Overflow, underflow and loss of accuracy are potential problems. `Rational`s that are too absolutely large or too absolutely small may overflow or underflow; some `Rational`s can not be represented exactly as a `Float`.)

9 $\mathbb{R} \to \mathbb{Q}$ — Approximation

This section describes functions that approximate a number (usually a `Float`) by a `Rational`. See §7 for approximation of a number by a `Rational` with a fixed denominator. See §17 for approximation by a `Rational` of a string representation of a real number.

9.1 Intervals

Some of the approximation functions return an `Interval`.

The file `rat_interval.q` is a basic implementation of interval arithmetic, and is not included in the default prelude. It is not intended to provide a complete implementation of interval arithmetic.

The notions of 'open' and 'closed' intervals are not distinguished. Infinite and half-infinite intervals are not specifically provided. Some operations and functions may be missing.

The most likely functions to be used are simply the 'deconstructors'; see §9.1.1.

9.1.1 Interval Constructors and 'Deconstructors'

`Interval`s are constructed with the function `interval`.

`interval`$^{(\mathcal{U})}$ (N_1, N_2) — given a pair of numbers $(Z_1 <= Z_2)$, this returns the `Interval` $Z_1..Z_2$. This is the inverse of `lo_up`.

Example 33 *Constructing an interval.*

```
==> def V = interval (3,8) ; V
interval (3,8)
```

`lower`$^{(\mathcal{U})}$ V — returns the infimum (roughly, minimum) of V.

`upper`$^{(\mathcal{U})}$ V — returns the supremum (roughly, maximum) of V.

`lo_up`$^{(\mathcal{U})}$ V — returns a pair (L, U) containing the lower L and upper D extrema of the interval V. This is the inverse of `interval` as defined on number pairs.

Example 34 *Deconstructing an interval.*

```
==> lower V; upper V
3
8
==> lo_up V
(3,8)
```

9.1.2 Interval Type Predicate and Tests

This type may be used as a type predicate in the usual way (`Q:Rational`).

`isexact`$^{(\mathcal{U})}$ V — returns whether an interval V has exact extrema.

`isinexact`$^{(\mathcal{U})}$ V — returns whether an interval V has an inexact extremum.

`isinterval`$^{(\mathcal{U})}$ X — returns whether X is of type `Interval`.

`isinterval_val`$^{(\mathcal{U})}$ X — returns whether X has an interval value.

`isratinterval_val`$^{(\mathcal{U})}$ X — returns whether X has an interval value with rational extrema.

`isintinterval_val`$^{(\mathcal{U})}$ X — returns whether X has an interval value with integral extrema.

Example 35 *Interval value tests.*

```
==> def L = [interval(0,1), interval(0,1%1), interval(0,3%2), interval(0,1.5)]
==> map isexact L
[true,true,true,false]
==> map isinexact L
[false,false,false,true]
==> map isinterval L
[true,true,true,true]
==> map isinterval_val L
[true,true,true,true]
==> map isratinterval_val L
[true,true,true,true]
==> map isintinterval_val L
[true,true,false,false]
```

9.1.3 Interval Arithmetic Operators and Relations

The standard arithmetic operators $(+)^{(\mathcal{U})}$, $(-)^{(\mathcal{U})}$, $(*)^{(\mathcal{U})}$, $(/)^{(\mathcal{U})}$ and $(\%)^{(\mathcal{U})}$ are overloaded for `Intervals`. The divide operators (/) and (%) do not produce a result if the right operand is an interval containing 0.

Example 36 *Some intervals.*

```
==> def A = interval (11, 19); def B = interval (16, 24)
==> def C = interval (21, 29); def D = interval (23, 27)
```

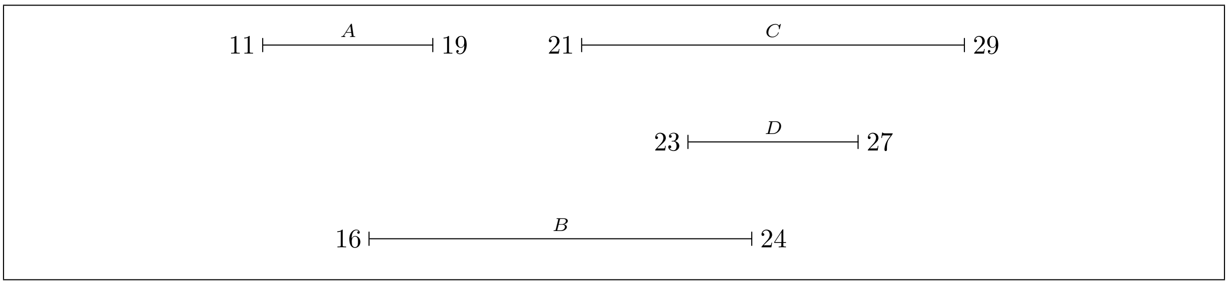

Figure 1: Some intervals.

Example 37 *Interval arithmetic.*

```
==> def P = interval (0, 1); def S = interval (-1, 1)
==> A + B
interval (27,43)
==> A - B
interval (-13,3)
==> A * B
interval (176,456)
==> P * 2
interval (0,2)
==> (-2) * P
interval (-2,0)
==> -C
interval (-29,-21)
==> S * A
interval (-19,19)
==> A % 2
interval (11%2,19%2)
==> A / 2
interval (5.5,9.5)
==> reciprocal A
interval (1%19,1%11)
==> 2 % A
interval (2%19,2%11)
==> A % B
interval (11%24,19%16)
==> A % A // notice that the intervals are mutually independent here
interval (11%19,19%11)
```

There are also some relations defined for `Intervals`. The standard relations $(=)^{(\mathcal{U})}$ and $(<>)^{(\mathcal{U})}$ are overloaded.

However, rather than overloading (<), (<=), (>), (>=), which could be used for either ordering or containment with some ambiguity, the module defines `(before)`, `(within)`, and so on.

'Strictness' refers to the properties at the end-points.

V_1 `before`$^{(\mathcal{U})}$ V_2 — returns whether V_1 is entirely before V_2.

V_1 `strictly_before`$^{(\mathcal{U})}$ V_2 — returns whether V_1 is strictly entirely before V_2.

V_1 `after`$^{(\mathcal{U})}$ V_2

V_1 `strictly_after`$^{(\mathcal{U})}$ V_2

V_1 `within`$^{(\mathcal{U})}$ V_2 — returns whether V_1 is entirely within V_2; i.e. whether V_1 is subinterval of V_2.

V_1 `strictly_within`$^{(\mathcal{U})}$ V_2 — returns whether V_1 is strictly entirely within V_2; i.e. whether V_1 is proper subinterval of V_2.

V_1 `without`$^{(\mathcal{U})}$ V_2 — returns whether V_1 entirely contains V_2; i.e. whether V_1 is superinterval of V_2. 'Without' is used in the sense of outside or around.

V_1 `strictly_without`$^{(\mathcal{U})}$ V_2 — returns whether V_1 strictly entirely contains V_2; i.e. whether V_1 is proper superinterval of V_2.

V_1 `disjoint`$^{(\mathcal{U})}$ V_2

V_1 `strictly_disjoint`$^{(\mathcal{U})}$ V_2

Example 38 *Interval relations.*

```
==> A = B
false
==> A = A
true
==> A before B
false
==> A before C
true
==> C before A
false
==> A disjoint B
false
==> C disjoint A
true
==> A within B
false
==> A within C
false
==> D within C
true
==> C within D
false
==> A strictly_within A
false
==> A within A
true
```

(The symbols `A` *through* `D` *were defined in example 36.)*

These may also be used with a simple (`Real`) value, and in particular to test membership.

Example 39 *Membership.*

```
==> 10 within A
false
==> 11 within A
true
```

```
==> 11.0 within A
true
==> 12 within A
true
==> 12.0 within A
true
==> 10 strictly_within A
false
==> 11 strictly_within A
false
==> (11%1) strictly_within A
false
==> 12 strictly_within A
true
==> (12%1) strictly_within A
true
```

(The symbol `A` *was defined in example 36.)*

9.1.4 Interval Maths

Some standard functions are overloaded for intervals; some new functions are provided.

`abs`$^{(\mathcal{U})}$ V — returns the interval representing the range of (X) as X varies over V.

Example 40 *Absolute interval.*

```
==> abs (interval (1, 5))
interval (1,5)
==> abs (interval (-1, 5))
interval (0,5)
==> abs (interval (-5, -1))
interval (1,5)
```

`sgn`$^{(\mathcal{U})}$ V — returns the interval representing the range of $\mathrm{sgn}(X)$ as X varies over V.

`size`$^{(\mathcal{U})}$ V — returns the length of an interval.

Example 41 *Absolute interval.*

```
==> size D
4
```

(The symbol `D` *was defined in example 36.)*

9.2 Least Complex Approximation within ε

`rational_approx_epsilon`$^{(\mathcal{U})}$ ε R — Find the least complex (see §5.1.3) `Rational` approximation to R (usually a `Float`) that is ε-close. That is find the Q with the smallest possible denominator such that such that $|Q - R| \leq \varepsilon$. $(\varepsilon > 0.)$

Example 42 *Rational approximation to* $\pi \approx 3.142 \approx 22/7$.

```
==> def Pi = 4 * atan 1
==> rational_approx_epsilon .01 Pi
22%7
==> abs (_ - Pi)
0.00126448926734968
```

Example 43 *The golden ratio* $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.618$.

```
==> def Phi = (1 + sqrt 5) / 2
==> rational_approx_epsilon .001 Phi
55%34
==> abs (_ - Phi)
0.000386929926365465
```

`rational_approxs_epsilon`$^{(u)}$ ε R — Produce a list of ever better `Rational` approximations to R (usually a `Float`) that is eventually ε-close. ($\varepsilon > 0$.)

Example 44 *Rational approximations to* π.

```
==> rational_approxs_epsilon .0001 Pi
[3%1,25%8,47%15,69%22,91%29,113%36,135%43,157%50,179%57,201%64,223%71,245%78,267
%85,289%92,311%99,333%106]
```

(The symbol `Pi` *was defined in example 42.)*

Example 45 *Rational approximations to the golden ratio* ϕ*; these approximations are always reverse consecutive Fibonacci numbers (from* F_1*:* $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$*).*

```
==> rational_approxs_epsilon .0001 Phi
[1%1,3%2,8%5,21%13,55%34,144%89]
```

(The symbol `Phi` *was defined in example 43.)*

`rational_interval_epsilon`$^{(u)}$ ε R — Find the least complex (see §5.1.3) `Rational` interval containing R (usually a `Float`) that is ε-small. That is find the least complex (see §5.1.3) $Q_1 \le Q_2$ such that $R \in [Q_1, Q_2]$ and $Q_2 - Q_1 \le \varepsilon$. ($\varepsilon > 0$.)

Example 46 *Rational interval surrounding* π.

```
==> def I_Pi = rational_interval_epsilon .01 Pi ; I_Pi
interval (47%15,22%7)
==> float (lower I_Pi) ; Pi ; float (upper I_Pi)
3.13333333333333
3.14159265358979
3.14285714285714
```

(The symbol `Pi` *was defined in example 42. The functions* `lower` *and* `upper` *are described in §9.1.1.)*

Example 47 *Rational interval surrounding the golden ratio* ϕ.

```
==> rational_interval_epsilon .001 Phi
interval (55%34,89%55)
==> size _
1%1870
```

(The symbol `Phi` *was defined in example 43. The function* `size` *is described in §9.1.4.)*

9.3 Best Approximation with Bounded Denominator

`rational_approx_max_den`$^{(u)}$ $MaxDen\ R$ — Find the closest `Rational` approximation to R (usually a `Float`) that has a denominator no greater than $MaxDen$. ($MaxDen > 0$)

Example 48 *Rational approximation to π.*

```
==> rational_approx_max_den 10 Pi
22%7
```

(The symbol `Pi` *was defined in example 42.)*

Example 49 *Rational approximation to the golden ratio ϕ.*

```
==> rational_approx_max_den 1000 Phi
1597%987
```

(The symbol `Phi` *was defined in example 43.)*

`rational_approxs_max_den`$^{(u)}$ $MaxDen\ R$ — Produce a list of ever better `Rational` approximations to R (usually a `Float`) while the denominator is bounded by $MaxDen$. ($MaxDen > 0$)

Example 50 *Rational approximations to π.*

```
==> rational_approxs_max_den 100 Pi
[3%1,25%8,47%15,69%22,91%29,113%36,135%43,157%50,179%57,201%64,223%71,245%78,267
%85,289%92,311%99]
```

(The symbol `Pi` *was defined in example 42.)*

Example 51 *Rational approximations to the golden ratio ϕ.*

```
==> rational_approxs_max_den 100 Phi
[1%1,3%2,8%5,21%13,55%34,144%89]
```

(The symbol `Phi` *was defined in example 43.)*

`rational_interval_max_den`$^{(u)}$ $MaxDen\ R$ — Find the smallest `Rational` interval containing R (usually a `Float`) that has endpoints with denominators no greater than $MaxDen$. ($MaxDen > 0$)

Example 52 *Rational interval surrounding π.*

```
==> def I_Pi = rational_interval_max_den 100 Pi ; I_Pi
interval (311%99,22%7)
==> float (lower I_Pi) ; Pi ; float (upper I_Pi)
3.14141414141414
3.14159265358979
3.14285714285714
```

(The symbol `Pi` *was defined in example 42.)*

Example 53 *Rational interval surrounding the golden ratio ϕ.*

```
==> rational_interval_max_den 1000 Phi
interval (987%610,1597%987)
```

(The symbol `Phi` *was defined in example 43.)*

To approximate for a particular (rather than bounded) denominator, use `round_to_multiple` instead (see §7).

10 Decomposition

There is more than one way to 'decompose' a rational number into its 'components'.

It might be split into an integer and a fraction part — see §6.2; or sign and absolute value — see §4.1; or numerator and denominator — see §2.2.

11 Continued Fractions

11.1 Introduction

In "Q+Q", a continued fraction

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cdots + \cfrac{1}{a_n}}}$$

where $\forall i > 0 \bullet a_i \neq 0$, is represented by $[a_0, a_1, a_2, \ldots, a_n]$.

A 'simple' continued fraction is one in which $\forall i \bullet a_i \in \mathbb{Z}$ and $\forall i > 0 \bullet a_i > 0$.

Simple continued fractions for rationals are not quite unique since $[a_0, a_1, \ldots, a_n, 1] = [a_0, a_1, \ldots, a_n + 1]$. We will refer to these as the 'non-standard' and 'standard' forms, respectively. The following functions return the standard form.

11.2 Generating Continued Fractions

11.2.1 Exact

`continued_fraction`$^{(\mathcal{U})}$ Q — Find 'the' (exact) continued fraction of a rational (including, trivially, integer) value Q.

Example 54 *The rational* $\frac{1234}{1001}$.

```
==> continued_fraction (1234%1001)
[1,4,3,2,1,1,1,8]
==> evaluate_continued_fraction _
1234%1001
```

11.2.2 Inexact

`continued_fraction_max_terms`$^{(\mathcal{U})}$ N R — Find up to N initial terms of continued fraction of the value R with the 'remainder', if any, in the final element. (If `continued_fraction_max_terms N R` returns a list of length N or less, then the result is exact.)

Example 55 *First 5 terms of the continued fraction for the golden ratio* ϕ.

```
==> continued_fraction_max_terms 5 Phi
[1,1,1,1,1,1.61803398874989]
==> evaluate_continued_fraction _
1.61803398874989
```

(The symbol `Phi` *was defined in example 43.)*

`continued_fraction_epsilon`$^{(\mathcal{U})}$ ε R — Find enough of the initial terms of a continued fraction to within ε of the value R with the 'remainder', if any, in the final element.

Example 56 *First few terms of the value* $\sqrt{2}$.

```
==> continued_fraction_epsilon .001 (sqrt 2)
[1,2,2,2,2,2.41421356237241]
==> map float (convergents _)
[1.0,1.5,1.4,1.41666666666667,1.41379310344828,1.41421356237309]
```

11.3 Evaluating Continued Fractions

`evaluate_continued_fraction`$^{(\mathcal{U})}$ AA — Fold a continued fraction AA into the value it represents. This function is not limited to simple continued fractions. (Exact simple continued fractions are folded into a rational.)

Example 57 *The continued fraction* $[1, 2, 3, 4]$ *and the non-standard form* $[4, 3, 2, 1]$.

```
==> evaluate_continued_fraction [1,2,3,4]
43%30
==> continued_fraction _
[1,2,3,4]
==> evaluate_continued_fraction [4,3,2,1]
43%10
==> continued_fraction _
[4,3,3]
```

11.3.1 Convergents

`convergents`$^{(\mathcal{U})}$ AA — Calculate the convergents of the continued fraction AA. This function is not limited to simple continued fractions.

Example 58 *Convergents of a continued fraction approximation of the value* $\sqrt{2}$.

```
==> continued_fraction_max_terms 5 (sqrt 2)
[1,2,2,2,2,2.41421356237241]
==> convergents _
[1%1,3%2,7%5,17%12,41%29,1.41421356237309]
```

12 Rational Complex Numbers

Q provides various types of number, including `Ints` ($\mathbb{Z}$), `Floats` ($\mathbb{R}$, roughly), `Complex` numbers ($\mathbb{C}$) and Gaussian integers ($\mathbb{Z}[i]$); the `rational.q` module adds `Rationals` ($\mathbb{Q}$) and rational complex numbers ($\mathbb{Q}[i]$).
Figures 2 and 3 illustrate the relationships between the types.

$$\begin{tikzcd}
 & & \mathbb{C} = \mathbb{R}[i] \arrow[dl, dash] \arrow[dr, dash] & \\
 & \mathbb{Q}[i] \arrow[dl, dash] \arrow[dr, dash] & & \mathbb{R} \arrow[dl, dash] \\
\mathbb{Z}[i] \arrow[dr, dash] & & \mathbb{Q} \arrow[dl, dash] & \\
 & \mathbb{Z} & &
\end{tikzcd}$$

Figure 2: Numbers.

Functions concerning 'complex numbers' are named with '`comp`', whereas functions concerning 'complexity' (see §5.1) are named with '`cplx`'.

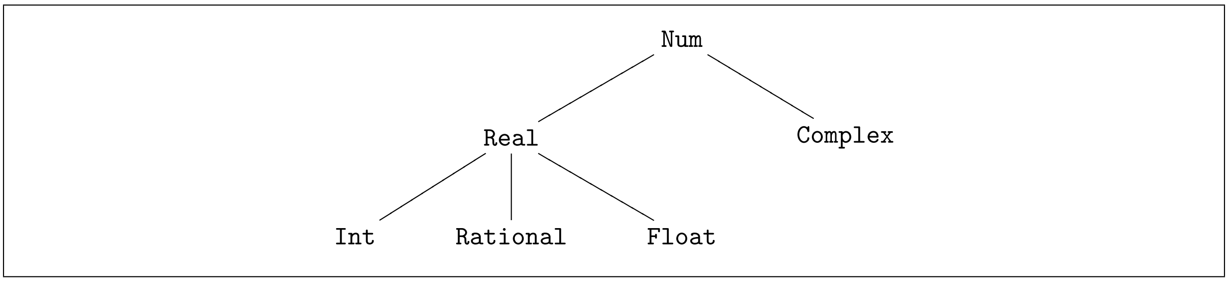

Figure 3: Types.

12.1 Rational Complex Constructors and 'Deconstructors'

Complex numbers can have rational parts.

Example 59 *Forming a rational complex.*

```
==> 1 + i * (1%2)
1%1+:1%2
==> _ * _
3%4+:1%1
```

And rational numbers can be given complex parts.

Example 60 *Complex rationals and complicated rationals.*

```
==> (1+2*i) % (3+4*i)
11%25+:2%25
==> _ * (3+4*i)
1%1+:2%1
==> ((4%1)*i) % 2
0%1+:2%1
==> ((4%1)*i) % (1%2)
0%1+:8%1
==> ((4%1)*i) % (1+(1%2)*i)
8%5+:16%5
==> _ * (1+(1%2)*i)
0%1+:4%1
==> ((4%1)*i) / (1+(1%2)*i)
1.6+:3.2
```

The various parts of a complex rational may be deconstructed using combinations of `num` and `den` and the standard functions `re` and `im`.

Thus, taking real and imaginary parts first, a rational complex number may be considered to be

$$\frac{X_n}{X_d} + \frac{Y_n}{Y_d} * \mathrm{i}$$

with $X_n, X_d, Y_n, Y_d \in \mathbb{Z}$.

A rational complex number may also be decomposed into its 'numerator' and 'denominator', where these are both integral complex numbers, or 'Gaussian integers', and the denominator is a minimal choice in some sense.

One way to do this is so that the denominator is the minimum positive integer. The denominator is a complex number with zero imaginary part.

Thus, taking numerator and denominator parts first, a rational complex number may be considered to be

$$\frac{N_x + N_y * \mathrm{i}}{D + 0 * \mathrm{i}}$$

with $N_x, N_y, D \in \mathbb{Z}$.

Another way to do this is so that the denominator is a Gaussian integer with minimal absolute value. Thus, taking numerator and denominator parts first, a rational complex number may be considered to be

$$\frac{N_x + N_y * \mathrm{i}}{D_x + D_y * \mathrm{i}}$$

with $N_x, N_y, D_x, D_y \in \mathbb{Z}$.
The D_x, D_y are not unique, but can be chosen such that $D_x > 0$ and either $|D_y| < D_x$ or $D_y = D_x > 0$.

`num_den_nat`$^{(\mathcal{U})}$ C — given a `Complex Rational` or `Int` C, returns a pair (N, D) containing an integral complex (Gaussian integral) numerator N, and the smallest natural (i.e. positive integral real) complex denominator D, i.e. a complex number where $\Re(D) \in \mathbb{Z}$, $\Re(D) > 0$, $\Im(D) = 0$; i.e. the numerator and denominator of one 'normalised' form of C.

This is an inverse (up to equivalence) of `rational` as defined on integral complex pairs (see §2.1).

`num_den_gauss`$^{(\mathcal{U})}$ C — given a `Complex Rational` or `Int` C, returns a pair (N, D) containing an integral complex (Gaussian integral) numerator N, and an absolutely smallest integral complex denominator D chosen s.t. $\Re(D), \Im(D) \in \mathbb{Z}$, $\Re(D) > 0$, and either $|\Im(D)| < \Re(D)$ or $\Im(D) = \Re(D) > 0$; i.e. the numerator and denominator of another 'normalised' form of C.

This is an inverse (up to equivalence) of `rational` as defined on integral complex pairs (see §2.1).

`num_den`$^{(\mathcal{U})}$ C — synonymous with `num_den_gauss`.

This is an inverse (up to equivalence) of `rational` as defined on integer pairs (see §2.1).

`num`$^{(\mathcal{U})}$ C — given a `Complex Rational` or `Int` C, returns just the numerator of the normalised form of C given by `num_den` C.

`den`$^{(\mathcal{U})}$ C — given a `Complex Rational` or `Int` C, returns just the denominator of the normalised form of C given by `num_den` C.

Example 61 *Rational complex number deconstruction.*

```
==> def CQ = (1+2*i)%(3+3*i); CQ
1%2+:1%6
==> (re CQ, im CQ)
(1%2,1%6)
==> (num . re) CQ
1
==> (den . re) CQ
2
==> (num . im) CQ
1
==> (den . im) CQ
6
==> def (N_nat,D_nat) = num_den_nat CQ; (N_nat, D_nat)
(3+:1,6+:0)
==> N_nat % D_nat
1%2+:1%6
==> abs D_nat
6.0
==> def (N,D) = num_den_gauss CQ; (N, D)
(1+:2,3+:3)
==> def (N,D) = num_den CQ; (N, D)
(1+:2,3+:3)
==> N % D
1%2+:1%6
==> abs D
4.24264068711928
```

```
==> (re . num) CQ
1
==> (im . num) CQ
2
==> (re . den) CQ //always > 0
3
==> (im . den) CQ //always <= (re.den)
3
```

12.2 Rational Complex Type and Value Tests

Beware that `isintcompval` and `isratcompval` may return `true` even if the value is of `Complex` type with `Float` parts. However, these functions may be combined with `isexact`.

`iscomplex` X — standard function; returns whether X is of `Complex` type.

`iscompval` X — standard function; returns whether X has a `Complex` value ($\in \mathbb{C} = \mathbb{R}[i]$).

`isratcompval`$^{(\mathcal{U})}$ X — returns whether X has a rational complex value ($\in \mathbb{Q}[i]$).

`isintcompval`$^{(\mathcal{U})}$ X — returns whether X has an integral complex value ($\in \mathbb{Z}[i]$), i.e. a Gaussian integer value.

Example 62 *Rational complex number value tests.*

```
==> def L = [9, 9%1, 9%2, 4.5, sqrt 2, 1+i, 1%2+i, 0.5+i, inf, nan]
==> map isexact L
[true,true,true,false,false,true,true,false,false,false]
==> map isinexact L
[false,false,false,true,true,false,false,true,true,true]
==> map iscomplex L
[false,false,false,false,false,true,true,true,false,false]
==> map iscompval L
[true,true,true,true,true,true,true,true,true,true]
==> map (lambda X (isexact X and iscompval X)) L // "has exact complex value"
[true,true,true,false,false,true,true,false,false,false]
==> map isratcompval L
[true,true,true,true,true,true,true,true,false,false]
==> map (lambda X (isexact X and isratcompval X)) L
[true,true,true,false,false,true,true,false,false,false]
==> map isintcompval L
[true,true,false,false,false,true,false,false,false,false]
==> map (lambda X (isexact X and isintcompval X)) L
[true,true,false,false,false,true,false,false,false,false]
==> map isratval L
[true,true,true,true,true,false,false,false,false,false]
==> map (lambda X (isexact X and isratval X)) L
[true,true,true,false,false,false,false,false,false,false]
==> map isintval L // for comparison
[true,true,false,false,false,false,false,false,false,false]
==> map (lambda X (isexact X and isintval X)) L
[true,true,false,false,false,false,false,false,false,false]
```

See §2.4 for some details of rational type and value tests.

12.3 Rational Complex Arithmetic Operators and Relations

The standard arithmetic operators (+), (−), (∗), (/), (%), (^), (=) and (<>) are overloaded to have at least one complex and/or rational operand, but (<), (<=), (>), (>=) are not, as complex numbers are unordered.

Example 63 *Rational complex arithmetic.*

```
==> def W = 1%2+3%4*i; def Z = 5%6+7%8*i
==> W + Z; W % Z
4%3+:13%8
618%841+:108%841
==> W / Z
0.734839476813318+:0.128418549346017
==> W ^ 2
-0.3125+:0.75
==> W = Z
false
==> W = W
true
```

12.4 Rational Complex Maths

The standard functions `re` and `im` work with rational complex numbers (see §12.1).
The standard functions `polar`, `abs` and `arg` work with rational complex numbers, but the results are inexact.

Example 64 *Rational complex maths.*

```
==> polar (1%2) (1%2)
0.438791280945186+:0.239712769302102
==> abs (4%2 + 3%2*i)
2.5
==> arg (-1%1)
3.14159265358979
```

There are some additional useful functions for calculating with rational complex numbers and more general mathematical values.

`norm_gauss`$^{(\mathcal{U})}$ C — returns the Gaussian norm $||C||$ of any `Complex` (or `Real`) number C; this is the square of the absolute value, and is returned as an (exact) `Int`.

`div_mod_gauss`$^{(\mathcal{U})}$ N D — performs Gaussian integer division, returning (Q, R) where Q is a (not always unique) quotient, and R is a (not always unique) remainder. Q and R are such that $N = Q*D+R$ and $||R|| < ||D||$ (equivalently, $|R| < |D|$).

N `div_gauss`$^{(\mathcal{U})}$ D — returns just a quotient from Gaussian integer division as produced by `div_mod_gauss` N D.

N `mod_gauss`$^{(\mathcal{U})}$ D — returns just a remainder from Gaussian integer division as produced by `div_mod_gauss` N D.

`gcd_gauss`$^{(\mathcal{U})}$ C_1 C_2 — returns a GCD G of the Gaussian integers C_1, C_2. This is chosen so that s.t. $\Re(G) > 0$, and either $|\Im(G)| < \Re(G)$ or $\Im(G) = \Re(G) > 0$;

`euclid_gcd`$^{(\mathcal{U})}$ *Zero Mod X Y* — returns a (non-unique) GCD calculated by performing the Euclidean algorithm on the values X and Y (of any type) where *Zero* is a predicate for equality to 0, and *Mod* is a binary modulus (remainder) function.

`euclid_alg`$^{(\mathcal{U})}$ *Zero Div X Y* — returns (G, A, B) where the G is a (non-unique) GCD and A, B are (arbitrary, non-unique) values such that $A*X+B*Y = G$ calculated by performing the generalised Euclidean algorithm on the values X and Y (of any type) where *Zero* is a predicate for equality to 0, and *Div* is a binary quotient function.

Example 65 *More rational complex and other maths.*

```
==> norm_gauss (1+3*i)
10
==> abs (1+3*i)
3.16227766016838
==> norm_gauss (-5)
25
==> def (Q,R) = div_mod_gauss 100 (12+5*i); (Q, R)
(7 +: -3,1+:1)
==> Q*(12+5*i) + R
100+:0
==> 100 div_gauss (12+5*i)
7 +: -3
==> 100 mod_gauss (12+5*i)
1+:1
==> div_mod_gauss 23 5
(5+:0,-2+:0)
==> gcd_gauss (1+2*i) (3+4*i)
1+:0
==> gcd_gauss 25 15
5+:0
==> euclid_gcd (=0) (mod_gauss) (1+2*i) (3+4*i)
1+:0
==> euclid_gcd (=0) (mod) 25 15
5
==> def (G,A,B) = euclid_alg (=0) (div_gauss) (1+2*i) (3+4*i); G; (A,B)
1+:0
(-2+:0,1+:0)
==> A*(1+2*i)+B*(3+4*i)
1+:0
==> def (G,A,B) = euclid_alg (=0) (div) 25 15; G; (A,B)
5
(-1,2)
==> A*25+B*15
5
```

12.5 Rational Complex Type Simplification

`comp_simplify`$^{(\mathcal{U})}$ C — returns Q with `Complex` numbers simplified to `Reals`, if possible.

`ratcomp_simplify`$^{(\mathcal{U})}$ C — returns Q with `Rationals` simplified to `Ints`, and `Complex` numbers simplified to `Reals`, if possible.

Example 66 *Rational complex number type simplification.*

```
==> def L = [9+i, 9%1+i, 9%2+i, 4.5+i, 9%1+0*i, 9%2+0*i, 4.5+0.0*i]; L
[9+:1,9%1+:1,9%2+:1,4.5+:1,9%1+:0,9%2+:0,4.5+:0.0]
==> map comp_simplify L
[9+:1,9%1+:1,9%2+:1,4.5+:1,9%1,9%2,4.5+:0.0]
==> map ratcomp_simplify L
[9+:1,9+:1,9%2+:1,4.5+:1,9,9%2,4.5+:0.0]
```

See §5.3 for some details of rational type simplification.

13 String Formatting and Evaluation

13.1 The Naming of the String Conversion Functions

There are several families of functions for converting between strings and `Rationals`.

The functions that convert from `Rational`s to strings have names based on that of the standard function `str`. The `str_*` functions convert to a formatted string, and depend on a 'format structure' parameter (see §13.2). The `strs_*` functions convert to a tuple of string fragments.

The functions that convert from strings to `Rational`s have names based on that of the standard function `val`. The `val_*` functions convert from a formatted string, and depend on a format structure parameter. The `sval_*` functions convert from a tuple of string fragments.

There are also `join_*` and `split_*` functions to join string fragments into formatted strings, and to split formatted strings into string fragments, respectively; these depend on a format structure parameter.

These functions are not always invertible, because some of the functions reduce an error term to just a sign, e.g. `str_real_approx_dp` may round a value. Thus sometimes the `join_*` and `split_*` pairs, and the `str_*` and `val_*` pairs are not quite mutual inverses.

This is illustrated in Figure 4; dotted lined indicate a possibly imperfect mapping.

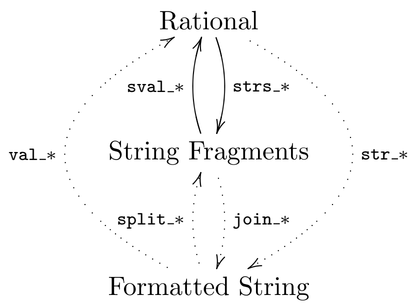

Figure 4: String functions.

13.2 Internationalisation and Format Structures

Many of the string formatting functions in the following sections are parameterised by a 'format structure'.

Throughout this document, the formal parameter for the format structure will be *Fmt*. This is simply a map from some string 'codes' to functions as follows. The functions are mostly from strings to a string, or from a string to a tuple of strings.

`"sm"`: a function mapping a sign and an unsigned mantissa (or integer) strings to a signed mantissa (or integer) string.

`"se"`: a function mapping a sign and an unsigned exponent string to a signed exponent string.

`"-s"`: a function mapping a signed number string to a pair containing a sign and the unsigned number string.

`"gi"`: a function mapping an integer representing the group size and an integer string to a grouped integer string

`"gf"`: a function mapping an integer representing the group size and a fraction-part string to a grouped fraction-part string

`"-g"`: a function mapping a grouped number string to an ungrouped number string

`"zi"`: a function mapping an integer number string to a number string. The input string representing zero integer part is `""`, which should be mapped to the desired representation of zero. All other number strings should be returned unaltered.

`"zf"`: a function mapping a fraction-part number string to a number string. The input string representing zero fraction part is `""`, which should be mapped to the desired representation of zero. All other number strings should be returned unaltered.

`"ir"`: a function mapping initial and recurring parts of a fraction part to the desired format.

"-ir": a function mapping a formatted fraction part to the component initial and recurring parts.

"if": a function mapping an integer string and fraction part string to the radix-point formatted string

"-if": a function mapping a radix-point formatted string to the component integer fraction part strings

"me": a function mapping a mantissa string and exponent string to the formatted exponential string

"-me": a function mapping a formatted exponential string to the component mantissa and exponent strings

"e": a function mapping an 'error' number (not string) and a number string to a formatted number string indicating the sign of the error.

"-e": a function mapping a formatted number string indicating the sign of the error to the component 'error' string (not number) and number strings.

Depending upon the format structure, some parameters of some of the functions taking a format structure may have no effect. For example, an *IntGroup* parameter specifying the size of the integer digit groups will have no effect if the integer group separator is the empty string.

`create_format`[(u)] *Options* is a function that provides an easy way to prepare a 'format structure' from the simpler 'options structure'.

The options structure is another dictionary, but from more descriptive strings to a string or tuple of strings.

For example, `format_uk`[(u)] is generated from `options_uk`[(u)] as follows:

```
public const var options_uk;
def options_uk =
  dict [
    ("sign", ("-","","")),              //alternative: ("-"," ","+")
    ("exponent sign", ("-","","")),     //alternative: ("-","","+")
    ("group separator", ","),           //might be " " or "." or "'" elsewhere
    ("zero", "0"),
    ("radix point", "."),               //might be "," elsewhere
    ("fraction group separator", ","),
    ("fraction zero", "0"),             //alternative: ""
    ("recur brackets", ("[","...]")),
    ("exponent", "*10^"),               //(poor) alternative: "e"
    ("error sign", ("-","","+")),
    ("error brackets", ("(",")")),
  ];

public const var format_uk;
def format_uk = create_format options_uk;
```

The exponent string need not depend on the radix, as the numerals for the number radix in that radix are always "10".

Beware of using "e" or "E" as an exponent string as these have the potential of being treated as digits in, e.g., hexadecimal.

Format structures do not have to be generated via `create_format`; they may also be constructed directly.

13.3 Digit Grouping

Some functions take *Group* parameters. A value of 0 means "don't group".

13.4 Radices

The functions that produce a decimal expansion take a `Radix` argument.

The fraction parts are expanded in that radix (or 'base'), in addition to the integer parts. The parameter `Radix` is not restricted to the usual $\{2, 8, 10, 16\}$, but may be any `Int` from 2 to 36; the numerals ('digits') are chosen from ["0", ..., "9", "A", ..., "Z"]. The letter-digits are always upper case.

The functions do not attach a prefix (such as `"0x"` for hexadecimal) to the resulting string.

13.5 Error Terms

Some functions return a value including an 'error' term (in a tuple) or sign (at the end of a string). Such an error is represents what the next digit would be as a fraction of the radix.

Example 67 *Error term in the tuple of string 'fragments'.*

```
==> strs_real_approx_sf 10 floor 3 (234567%100000)
("+","2","34",567%1000)
==> strs_real_approx_sf 10 ceil 3 (234567%100000)
("+","2","35",-433%1000)
```

(The function `strs_real_approx_sf` *is described in §16.1.2.)*

In strings, only the sign of the error term is given. A "`+`" should be read as "and a bit more"; "`-`" as "but a bit less".

Example 68 *Error sign in the string.*

```
==> str_real_approx_sf format_uk 10 0 0 floor 3 (234567%100000)
"2.34(+)"
==> str_real_approx_sf format_uk 10 0 0 ceil 3 (234567%100000)
"2.35(-)"
```

(The function `str_real_approx_sf` *is described in §16.1.2.)*

14 $\mathbb{Q} \leftrightarrow$ Fraction String ("$I + N/D$")

14.1 Formatting to Fraction Strings

`str_vulgar`$^{(\mathcal{U})}$ Q — returns a `String` representing the `Rational` (or `Int`) Q in the form

- "$[-]N/D$"

`str_vulgar_or_int`$^{(\mathcal{U})}$ Q — returns a `String` representing the `Rational` (or `Int`) Q in one of the forms

- "$[-]N/D$"
- "[−]I"

`str_mixed`$^{(\mathcal{U})}$ Q — returns a `String` representing the `Rational` (or `Int`) Q in one of the forms

- "$I + N/D$"
- "$-(I + N/D)$"
- "$[-]N/D$"
- "$[-]I$"

Example 69 *The fraction string representations.*

```
==> def L = while (<= 3%2) (+(1%2)) (-3%2); L
[-3%2,-1%1,-1%2,0%1,1%2,1%1,3%2]
==> map str_vulgar L
["-3/2","-1/1","-1/2","0/1","1/2","1/1","3/2"]
==> map str_vulgar_or_int L
["-3/2","-1","-1/2","0","1/2","1","3/2"]
==> map str_mixed L
["-(1+1/2)","-1","-1/2","0","1/2","1","1+1/2"]
```

These might be compared to the behaviour of the standard function `str`.

`str` X — returns a `String` representing the value X.

Example 70 *The standard function* `str`*.*

```
==> map str L
["-3%2","-1%1","-1%2","0%1","1%2","1%1","3%2"]
```

14.2 Evaluation of Fraction Strings

`val_vulgar`$^{(\mathcal{U})}$ Str — returns a `Rational` Q represented by the `String` Str in the form

- "$[-]N/D$"

Such strings can also be evaluated by the `val_mixed` function.

`val_mixed`$^{(\mathcal{U})}$ Str — returns a `Rational` Q represented by the `String` Str

- "$I + N/D$"
- "$-(I + N/D)$"
- "$[-]N/D$" — thus `val_mixed` strictly extends `val_vulgar`
- "$[-]I$"

Example 71 *Evaluating fraction strings.*

```
==> val_vulgar "-22/7"
-22%7
==> val_mixed "1+5/6"
11%6
```

These might be compared to the behaviour of the standard function `val`.

`val` S — evaluates the `String` S.

Example 72 *The standard function* `val`*.*

```
==> val "1+5%6"
11%6
==> val "1+5/6"
1.83333333333333
```

15 $\mathbb{Q} \leftrightarrow$ Recurring Numeral Expansion String ("$I.F\overline{R}$")

See §13.2 for information about the formatting structure to be supplied in the Fmt parameter.

15.1 Formatting to Recurring Expansion Strings

`str_real_recur`$^{(\mathcal{U})}$ *Fmt Radix IntGroup Q* — returns a `String` (exactly) representing the `Rational` (or `Int`) Q as base-*Radix* expansion of one the forms

- "[−]int.frac"
- "[−]int.init_frac_part[smallest_recurring_frac_part. . .]"

Note that there is no *FracGroup* parameter.

Beware that the string returned by this function can be very long. The length of the recurring part of such a decimal expansion may be up to one less than the simplest denominator of Q.

Example 73 *The recurring radix expansion-type string representations.*

```
==> str_real_recur format_uk 10 3 (4000001%4) // grouped with commas
"1,000,000.25"
==> str_real_recur format_uk 10 0 (4000001%4) // no grouping
"1000000.25"
==> str_real_recur format_uk 10 3 (1000000%3)
"333,333.[3...]"
==> str_real_recur format_uk 10 3 (1000000%7)
"142,857.[142857...]"
==> str_real_recur format_uk 10 3 (-1%700)
"-0.00[142857...]"
==> str_real_recur format_uk 10 3 (127%128)
"0.9921875"
==> str_real_recur format_uk 2 4 (-127%128)
"-0.1111111"
==> str_real_recur format_uk 16 4 (127%128)
"0.FE"
==> str_real_recur format_uk 10 0 (70057%350) // 1%7 + 10001%50
"200.16[285714...]"
```

The function allows expansion to different radices (bases).

Example 74 *The recurring radix expansion in decimal and hexadecimal.*

```
==> str_real_recur format_uk 10 0 (1%100)
"0.01"
==> str_real_recur format_uk 16 0 (1%100)
"0.0[28F5C...]"
```

Example 75 *The recurring radix expansion in duodecimal.*

```
==> str_real_recur format_uk 12 0 (1%100)
"0.0[15343A0B62A68781B059...]"
```

Note that this bracket notation is not standard in the literature. Usually the recurring numerals are indicated by a single dot over the initial and final numerals of the recurring part, or an overline over the recurring part. For example $1/70 = 0.0\dot{1}4285\dot{7} = 0.0\overline{142857}$ and $1/3 = 0.\dot{3} = 0.\overline{3}$.

`strs_real_recur`$^{(\mathcal{U})}$ *Radix Q* — returns a quadruple of the four strings:

- the sign,
- integer part (which is empty for 0),
- initial fraction part
- and recurring fraction part (either and both of which may be empty).

Example 76 *The recurring radix expansion in decimal — the fragments.*

```
==> strs_real_recur 10 (100%7)
("+","14","","285714")
==> strs_real_recur 10 (-1%700)
("-","","00","142857")
==> strs_real_recur 10 (70057%350)
("+","200","16","285714")
```

This function may be used to also, e.g. format the integer part with comma-separated groupings.

`join_str_real_recur`$^{(u)}$ *Fmt IntGroup Sign I FracInit FracRecur* — formats the parts in the quadruple returned by `strs_real_recur` to the sort of string as returned by `str_real_recur`.

15.2 Evaluation of Recurring Expansion Strings

The `str_*` and `val_*` functions depend on a 'format structure' parameter (*Fmt*) such as `format_uk`. Conversions may be performed between `Rational`s and differently formatted strings if a suitable alternative format structure is supplied. See §13.2 for information about formatting structures.

`val_real_recur`$^{(u)}$ *Fmt Radix Str* — returns the `Rational` Q represented by the base-*Radix* expansion `String` *Str* of one the forms

- "[−]int.frac"
- "[−]int.init_frac_part[recurring_frac_part...]"

Example 77 *Conversion from the recurring radix expansion-type string representations.*

```
==> val_real_recur format_uk 10 "-12.345"
-2469%200
==> val_real_recur format_uk 10 "0.3"
3%10
==> val_real_recur format_uk 10 "0.[3...]"
1%3
==> val_real_recur format_uk 10 ".333[33...]"
1%3
==> val_real_recur format_uk 10 ".[9...]"
1%1
```

`sval_real_recur`$^{(u)}$ *Radix Sign IStr FracStr RecurPartStr* — returns the `Rational` Q represented by the parts

- sign
- integer part
- initial fraction part
- recurring fraction part

`split_str_real_recur`$^{(u)}$ *Fmt Str* — returns a tuple containing the parts

- sign
- integer part
- initial fraction part
- recurring fraction part

of one the forms

- "[−]int.frac"
- "[−]int.init_frac_part[recurring_frac_part...]"

16 $\mathbb{Q} \leftrightarrow$ Numeral Expansion String ("$I.F \times 10^E$")

See §13.2 for information about the formatting structure to be supplied in the *Fmt* parameter.

The exponent string `"*10^"` need not depend on the radix, as the numerals for the number radix in that radix are always `"10"`.

16.1 Formatting to Expansion Strings

16.1.1 Functions for Fixed Decimal Places

`str_real_approx_dp`$^{(u)}$ *Fmt Radix IntGroup FracGroup Round DP Q* — returns a string representing a numeral expansion approximation of *Q* to *DP* decimal places, using rounding mode *Round* (see §6.1).

Round is usually `round` or `round_unbiased`.

(*DP* may be positive, zero or negative; non-positive DPs may look misleading — use e.g. scientific notation instead.)

Example 78 *Decimal places.*

```
==> str_real_approx_dp format_uk 10 3 3 round 2 (22%7)
"3.14(+)"
==> str_real_approx_dp format_uk 10 3 3 ceil 2 (22%7)
"3.15(-)"
```

`strs_real_approx_dp`$^{(u)}$ *Radix Round DP Q* — returns a tuple of strings

- sign
- integer part
- fraction part

representing an expansion to a number of decimal places, together with

- the rounding "error": a fraction representing the next numerals

Example 79 *Decimal places — the fragments.*

```
==> strs_real_approx_dp 10 round 2 (22%7)
("+","3","14",2%7)
==> strs_real_approx_dp 10 ceil 2 (22%7)
("+","3","15",-5%7)
```

`join_str_real_approx`$^{(u)}$ *Fmt IntGroup FracGroup Sign I Frac Err* — formats the parts in the quadruple returned by `strs_real_approx_dp` or `strs_real_approx_sf` to the sort of string as returned by `str_real_approx_dp` or `str_real_approx_sf`.

16.1.2 Functions for Significant Figures

`str_real_approx_sf`$^{(u)}$ *Fmt Radix IntGroup FracGroup Round SF Q* — returns a string representing a numeral expansion approximation of *Q* to *SF* significant figures, using rounding mode *Round* (see §6.1).

Round is usually `round` or `round_unbiased`.

(*SF* must be positive.)

Example 80 *Significant figures.*

```
==> str_real_approx_sf format_uk 10 3 3 floor 2 (22%7)
"3.1(+)"
==> str_real_approx_sf format_uk 10 3 3 floor 2 ((-22)%7)
"-3.2(+)"
```

`strs_real_approx_sf`$^{(u)}$ *Radix Round SF Q* — returns a tuple of strings

- sign,
- integer part,
- fraction part,

representing an expansion to a number of significant figures, together with

- the rounding "error": a fraction representing the next numerals

`join_str_real_approx`$^{(u)}$ — see §16.1.1.

16.1.3 Functions for Scientific Notation and Engineering Notation

`str_real_approx_sci`$^{(u)}$ *Fmt Radix IntGroup FracGroup Round SF Q* — returns a string expansion with a number of significant figures in scientific notation, using rounding mode *Round* (see §6.1).

(*SF* must be positive; *ExpStep* is usually 3, *Radix* is usually 10, *Round* is usually `round` or `round_unbiased`; `str_real_approx_sci` is equivalent to `str_real_approx_eng` (below) with *ExpStep* = 1.)

`strs_real_approx_sci`$^{(u)}$ *Radix Round SF Q* — returns a tuple of strings:

- sign of mantissa,
- integer part of mantissa,
- fraction part of mantissa,
- sign of exponent,
- exponent magnitude

representing an expansion to a number of significant figures in scientific notation together with

- the rounding "error": a fraction representing the next numerals

`str_real_approx_eng`$^{(u)}$ *Fmt ExpStep Radix IntGroup FracGroup Round SF Q* — returns a string expansion with a number of significant figures in engineering notation, using rounding mode *Round*.

The *ExpStep* parameter specifies the granularity of the exponent; specifically, the exponent will always be divisible by *ExpStep*.

(*SF* must be positive; *ExpStep* is usually 3 and must be positive, *Radix* is usually 10, *Round* is usually `round` or `round_unbiased`.)

Example 81 *Engineering notation.*

```
==> str_real_approx_eng format_uk 3 10 3 3 round 7 (rational 999950)
"999.950,0*10^3"
==> str_real_approx_eng format_uk 3 10 3 3 round 4 999950
"1.000*10^6(-)"
```

`strs_real_approx_eng`$^{(u)}$ *ExpStep Radix Round SF Q* — returns a tuple of strings:

- sign of mantissa,
- integer part of mantissa,
- fraction part of mantissa,

- sign of exponent,
- exponent magnitude

representing an expansion to a number of significant figures in engineering notation together with

- the rounding "error": a fraction representing the next numerals

Example 82 *Engineering notation — the fragments.*

```
==> strs_real_approx_eng 3 10 round 7 (rational 999950)
("+","999","9500","+","3",0%1)
==> strs_real_approx_eng 3 10 round 4 999950
("+","1","000","+","6",-1%20)
```

`join_str_real_eng`$^{(u)}$ *Fmt IntGroup FracGroup MantSign MantI MantFrac ExpSign ExpI Err* — formats the parts in the quadruple returned by `strs_real_approx_eng` or `strs_real_approx_sci` to the sort of string as returned by `str_real_approx_eng` or `str_real_approx_sci`.

16.2 Evaluation of Expansion Strings

The `str_*` and `val_*` functions depend on a 'format structure' parameter (*Fmt*) such as `format_uk`. Conversions may be performed between `Rational`s and differently formatted strings if a suitable alternative format structure is supplied. See §13.2 for information about formatting structures.

`val_real_eng`$^{(u)}$ *Fmt Radix Str* — returns the `Rational` Q represented by the base-*Radix* expansion `String` *Str* of one the forms

- "[$-$]*int.frac*"
- "[$-$]*int.frac*e[$-$]*exponent*"

Example 83 *Conversion from the recurring radix expansion-type string representations.*

```
==> val_real_eng format_uk 10 "-12.345"
-2469%200
==> val_real_eng format_uk 10 "-12.345*10^2"
-2469%2
```

`sval_real_eng`$^{(u)}$ *Radix SignStr MantIStr MantFracStr ExpSignStr ExpStr* — returns the `Rational` Q represented by the parts

- sign
- integer part of mantissa
- fraction part of mantissa
- sign of exponent
- exponent

`split_str_real_eng`$^{(u)}$ *Fmt Str* — returns a tuple containing the string parts

- sign
- integer part of mantissa
- fraction part of mantissa
- sign of exponent
- exponent
- the "error" sign

of one the forms

- "[$-$]int.frac"
- "[$-$]int.frac $\times 10$^[$-$]exponent"

These functions can deal with the fixed decimal places, the significant figures and the scientific notation in addition to the engineering notation.

17 Numeral String → $\mathbb{Q}$ — Approximation

This section describes functions to approximate by a `Rational` a real number represented by a string. See §9 for approximation by a `Rational` of a `Float`.

The `str_*` and `val_*` functions depend on a 'format structure' parameter (*Fmt*) such as `format_uk`. Conversions may be performed between `Rationals` and differently formatted strings if a suitable alternative format structure is supplied. See §13.2 for information about formatting structures.

`val_eng_approx_epsilon`$^{(u)}$ *Fmt Radix Epsilon Str* — Find the least complex `Rational` approximation Q to the number represented by the base-*Radix* expansion `String` *Str* in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10^[−]exponent"

that is ε-close. That is find a Q such that $|Q - \text{val } Str| \leq \varepsilon$.

Example 84 `Rational` *from a long string.*

```
==> def Str = "123.456,789,876,543,212,345,678,987,654,321*10^27"
==> def X = val_real_eng format_uk 10 Str ; X
123456789876543212345678987654321%1000
==> def Q = val_eng_approx_epsilon format_uk 10 (1%100) Str; Q
1975308638024691397530863802469%16
==> float (X - Q)
0.0085
==> str_real_approx_eng format_uk 3 10 3 3 round 30 Q
"123.456,789,876,543,212,345,678,987,654*10^27(+)"
==> str_real_approx_eng format_uk 3 10 3 3 round 42 Q
"123.456,789,876,543,212,345,678,987,654,312,500,000,000*10^27"
==> float Q
1.23456789876543e+029
```

`val_eng_interval_epsilon`$^{(u)}$ *Fmt Radix Epsilon Str* — Find the least complex `Rational` interval containing the number represented by the base-*Radix* expansion `String` *Str* in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10^[−]exponent"

that is ε-small.

`val_eng_approx_max_den`$^{(u)}$ *Fmt Radix MaxDen Str* — Find the closest `Rational` approximation to the number represented by the base-*Radix* expansion `String` *Str* in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10^[−]exponent"

that has a denominator no greater than $MaxDen$. ($MaxDen > 0$)

`val_eng_interval_max_den`$^{(u)}$ *Fmt Radix MaxDen Str* — Find the smallest `Rational` interval containing the number represented by the base-*Radix* expansion `String` *Str* in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10^[−]exponent"

that has endpoints with denominators no greater than $MaxDen$. ($MaxDen > 0$)

Example 85 *Other* `Rational`*s from a long string.*

```
==> val_eng_approx_epsilon format_uk 10 (1%100) Str
1975308638024691397530863802469%16
==> val_eng_interval_epsilon format_uk 10 (1%100) Str
interval (3086419746913580308641974691358%25,3456790116543209945679011654321%28)
==> val_eng_approx_max_den format_uk 10 100 Str
9999999980000000199999998000000%81
==> val_eng_interval_max_den format_uk 10 100 Str
interval (9999999980000000199999998000000%81,3456790116543209945679011654321%28)
```